Another formal specification language

Saaman, Erik Harald

5. IMPLICIT FUNCTIONS

Abstract

In object-oriented methods for system modeling, or programming, the organization of datatypes in subclass hierarchies plays a central role in the structuring of models. The use of inheritance simplifies the construction, maintenance, and reuse of conceptual models. Since this is also a point of concern for formal specification, which is a form of system modeling, it seems worthwhile to incorporate an inheritance mechanism in a specification language.

In AFSL inheritance is realized by allowing implicit functions. Some example specifications are given that use this form of inheritance in a number of different ways. The examples are followed by a discussion of the ambiguity problems that can be caused by inheritance. A mechanism is introduced that can resolve many of these ambiguities.

5.1 Inheritance

The key aspect of the subclass relationship is that a "class" c_1 "inherits" all "properties" of any of its superclasses c_2, meaning that any operation defined for objects of class c_2 can also be applied to objects of class c_1. In AFSL inheritance is realized by allowing *implicit functions*, which are unary functions that are declared with the attribute `IMPLICIT`. An implicit function $f : s_1 \rightarrow s_2$ can be applied to the argument of a function $g : s_2 \rightarrow s_3$ without actually showing f. That is, for term $t : s_1$ the application $g(t)$ can be used as shorthand for $g(f(t))$. Here the effect of f being implicit is that s_1 inherits property g from s_2.

Implicit functions are also known as *coercions* (Cardelli & Wegner 1985). Implicit functions are similar to *injections* in ASF+SDF (Klint 1993), except that these injections are syntax-less, that is, they are denoted by an invisible name. The terminology "implicit function" is used in this thesis instead of "injection" because implicit functions do not need to be injective in the mathematical sense (that is, mapping unequal arguments to unequal results).

It is not the intention of AFSL to be an object-oriented language (according to whatever definition). Apart from implicit functions, there are no other language con-

structs that specifically facilitate object-oriented modeling. On the other hand, implicit functions can be used in situations where traditional object-oriented inheritance is not applicable, such as inheritance for has-a relationships (see the examples given in this chapter). In object-oriented methods objects usually have an internal state. For a proper modeling of states in AFSL so-called implicit lifting is used, which is not introduced until Chapter 6. Therefore, the given examples do not use state; in that respect the examples are atypical for object-oriented models.

There are no special restrictions on the possible values of an implicit function; implicit functions are in every respect ordinary functions, except that they can be used implicitly. We write $s_1 < s_2$ if and only if there is at least one implicit function $f : s_1 \to s_2$. Sort s_1 is said to be an *inheritor* of s_2 if and only if $s_1 <^* s_2$, where $<^*$ is the reflexive transitive closure of $<$. The terminology "s_1 is an inheritor of s_2" is used here instead of "s_1 is subsort of s_2" because the word "subsort" suggests that s_1 represents a subset of s_2, which does not need to be the case.

Making implicit functions in a term explicit is part of the expansion mechanism that was introduced in Section 3.8. That is, $t \twoheadrightarrow t'$ denotes that term t can be expanded to term t' by completing names and adding implicit functions, for example:

$$g(t) \twoheadrightarrow g(\underline{f}(t))$$

For clarity, added implicit functions are sometimes underlined. For simplicity, in examples the names in an expansion are often abbreviated. An implicit function that is inserted in a expansion is called a *conversion* (f in the example). The formal (re)definition of $\twoheadrightarrow$ is given in Section 5.8.

5.2 Example: List Notation

AFSL does not have a builtin sort for lists, therefore a generic specification of lists was given in 4.3. This self made definition of list does not allow the usual notation for lists (such as `[1,2,3]`) many languages have builtin. Therefore, a minor inconvenience is that each list has to be terminated by the constant `Empty` (such as `1&2&3&Empty`). This can be remedied by adding to the definition of lists (Figure 4.4) an implicit function that maps elements `x` of `XS` to the singleton list containing only `x`:

```
FUNC   Singleton : XS -> ListS[XS] (IMPLICIT)
AXIOM  Singleton x = x&Empty
```

Now `1&2&3` can be used as shorthand for `1&2&3&Empty`.

Here the implicit function `Singleton` is injective. Therefore, `XS` is a real subsort of `ListS[XS]` in the sense that `XS` is "included" in `ListS[XS]`. A generic implicit function for subsorts is defined in `SubsortM` (Figure 5.1). Using this module it is clearer to define the implicit function from elements to singleton lists by:

```
MODULE SubsortM [XS, YS]

VAR    x : XS
VAR    y : YS

SORT   XS
SORT   YS

IMPORT LogicM % Fig. 3.3 page 43

FUNC   Inject : XS -> YS (IMPLICIT)
REQ    x1 /= x2  ==>  Inject x1 /= Inject x2

END MODULE
```

Fig. 5.1: Specification of subsort relationship.

```
IMPORT SubsortM [XS, ListS[XS]]
AXIOM  Inject x = x&Empty
```

Note that `Inject` is not a parameter of module `SubsortM` because that would require the definition of an "inject" function for each import of `SubsortM`.

5.3 Example: Aliases

Sort names with nested indexes can become lengthy and barely descriptive. For example, if information about persons is stored as a list of pairs of type:

```
ListS[PairS[PersonS,PersonInfoS]]
```

it can be useful to rename this sort to `PersonTableS`, but AFSL does not have a renaming mechanism. An earlier version of AFSL had a sort definition construct that allowed `PersonTableS` to be declared as an alias of the nested name. Aliases where used frequently in the FAN case study.

If implicit functions are available no special language construct for sort definitions is needed. Instead, two sorts s_1 and s_2 can be semi-identified by declaring that they are each others subsorts. This is done in the generic module `AliasM` in Figure 5.2. Now `PersonTableS` can be defined to be an alias by:

```
SORT   PersonTableS
IMPORT AliasM [PersonTableS, ListS[PairS[PersonS,PersonInfoS]]]
```

Note that the axioms of `AliasM` are ambiguous since implicit functions can be added in numerous ways, for example:

```
MODULE AliasM [XS, YS]

VAR    x : XS
VAR    y : YS

SORT   XS
SORT   YS

IMPORT SubsortM [XS, YS] % Fig. 5.1 page 87
IMPORT SubsortM [YS, XS] % Fig. 5.1 page 87

AXIOM  Inject Inject y = y
AXIOM  Inject Inject x = x

END MODULE
```

Fig. 5.2: Specification of alias relationship.

```
AXIOM  Inject Inject Inject y = Inject y
AXIOM  Inject Inject y        = Inject Inject y
```

Resolving inheritance ambiguities like this is discussed in Section 5.7.

5.4 Example: Shapes

A typical object-oriented example model is the description of different types of geometrical shapes, which here are two-dimensional objects placed in an imaginary three-dimensional space (for example, a window in a graphical user-interface). A simplified model is given which serves to demonstrate the use of inheritance. Shapes (Figure 5.4) have a position and an area, no more features are assumed for shapes in general. The position of a shape is a three-dimensional coordinate (Figure 5.3). De z-coordinate of the position is the depth at which the shape is placed in space. The components of a coordinate and the area are assumed to be floating point numbers for simplicity, but should include a unit of measurement (such as “meters” and “square meters”) to be precise. Three special forms of shapes are specified: circles (Figure 5.5), with an additional radius feature; rectangles (Figure 5.6), with height and width features; and squares, which are rectangles that have a height equal to their width (Figure 5.7).

The sorts in this example are not defined inductively; instead they are specified by listing a number of key features which fully determine the observable properties of its elements. Here a *feature* of sort s is a function whose first domain type is s. The *key features* of s are the features of s that together determine the equality of s.

```
MODULE CoordinateM

IMPORT FloatM % Fig. 5.10 page 96

SORT   CoordS
FUNC   XCoord : CoordS -> FloatS
FUNC   YCoord : CoordS -> FloatS
FUNC   ZCoord : CoordS -> FloatS

%... further definition of functions for CoordS

END MODULE
```

Fig. 5.3: Specification of coordinates in three dimensional space.

```
MODULE ShapeM

VAR    sh : ShapeS

IMPORT LogicM      % Fig. 3.3 page 43
IMPORT CoordinateM % Fig. 5.3 page 89

SORT   ShapeS
FUNC   Position : ShapeS -> CoordS (IMPLICIT)
FUNC   Area     : ShapeS -> FloatS
AXIOM  Position sh1 = Position sh2  And
       Area sh1 = Area sh2
       ==>
       sh1 = sh2

FUNC   InFrontOf : ShapeS, ShapeS -> BoolS
AXIOM  sh1 InFrontOf sh2  <=>  ZCoord sh1 <= ZCoord sh2

FUNC   BiggerThan : ShapeS, ShapeS -> BoolS
AXIOM  sh1 BiggerThan sh2  <=>  Area sh1 >= Area sh2

%... Other operations for shapes

END MODULE
```

Fig. 5.4: Specification of shapes.

```
MODULE CircleM

VAR    circ : CircleS

IMPORT ShapeM % Fig. 5.4 page 89

SORT   CircleS
FUNC   Shape  : CircleS -> ShapeS (IMPLICIT)
FUNC   Radius : CircleS -> FloatS

AXIOM  Shape circ1 = Shape circ2  And
       Radius circ1 = Radius circ2
       ==>
       circ1 = circ2

AXIOM  Area circ = Pi * Radius circ * Radius circ

%... definition of other functions for circles

END MODULE
```

Fig. 5.5: Specification of circles.

For example, the key features of `ShapeS` are `Position` and `Area`. That is, shapes with the same area and position cannot be distinguished as elements of `ShapeS`. In fact, shapes can be viewed as records with fields `Position` and `Area`.

`ShapeS` is an abstract sort in the sense that it has derived features (`Area`) that cannot be defined for shapes in general, but that can be defined for some of its inheritors (`CircleS` and `RectangleS`). In this respect `Area` behaves like what is sometimes called a virtual method in object-oriented programming.

It can be argued that in a true object-oriented specification the equality axioms should be omitted since, for example, a rectangle is a shape and two rectangles are not equal even when they have the same position and area. It might even be true that any definition of equality for shapes is not needed and therefore over-specific. However, defining equality for any sort s has the advantage that it makes clear right away which properties of the elements of s are relevant for the specification, even if the equality itself is not used. For example, no matter how they are defined, `InFrontOf` and `Bigger` at most depend on the position and area of their arguments.

`Position` is an implicit function so that the individual coordinates of (the position of) a shape can be referred to directly using one of the coordinate selection functions, such as `ZCoord`. The resulting inheritance by `ShapeS` from `CoordS` is used in the axiom that defines the relation `InFrontOf` in terms of the z-coordinate of

```
MODULE RectangleM

VAR    rect : RectangleS

IMPORT ShapeM % Fig. 5.4 page 89

SORT   RectangleS
FUNC   Shape  : RectangleS -> ShapeS (IMPLICIT)
FUNC   Height : RectangleS -> FloatS
FUNC   Width  : RectangleS -> FloatS

AXIOM  Shape rect1  = Shape rect2   And
       Height rect1 = Height rect2  And
       Width rect1  = Width rect2
       ==>
       rect1 = rect2

AXIOM  Area rect = Height rect * Width rect

%... Operations for rectangles

END MODULE
```

Fig. 5.6: Specification of rectangles.

```
MODULE SquareM

VAR    sq   : SquareS

IMPORT RectangleM % Fig. 5.6 page 91

SORT   SquareS
FUNC   Rectangle : SquareS -> RectangleS (IMPLICIT)

AXIOM  Rectangle sq1 = Rectangle sq2
       ==>
       sq1 = sq2

AXIOM  Height sq = Width sq
LEMMA  Area sq = (Height sq) * (Height sq)

%... Operations for squares

END MODULE
```

Fig. 5.7: Specification of squares.

shapes. This axiom is shorthand for the expansion:

```
AXIOM sh1 InFrontOf sh2
      <=> ZCoord Position sh1 <= ZCoord Position sh2
```

`CircleS` is made an inheritor of `ShapeS` by the implicit function `Shape`. One key feature `Radius` is added to circles as an extension of shapes. Note that `Position` and `Area` are indirectly key features of `Circle` because `Shape` is a key feature. The dependency between the area and the radius of a circle is laid down in an axiom that uses inheritance by `CircleS` from `ShapeS`, it is shorthand for the expansion:

```
AXIOM Area Shape circ = Pi * Radius circ * Radius circ
```

In a similar way `RectangleS` is an inheritor of `ShapeS` and `SquareS` of `RectangleS`. For squares no additional key features are added to its rectangle features, only the possible values of `Height` and `Width` are restricted. Inheritance is used in the definition of `Area` for rectangles, in the axiom that restricts squares to rectangles with a height equal to their width, and in the lemma for squares.

The inheritance from coordinates to shapes through `Position` is conceptually different from the inheritance from shapes to circles through `Shape`. The first case corresponds to a so-called *has-a relationship* (a shape has a position) and the second to an *is-a relationship* (a circle is a shape). Technically, in AFSL there is no difference between these two kinds of inheritance. This is where the implicit function

approach diverges from inheritance in object-oriented methods which treat is-a and has-a differently (it is common that there is no inheritance for has-a relationships).

5.5 Example: Numbers

The next example demonstrates forms of inheritance that cannot be achieved in common object-oriented languages, but are quite common in programming languages. Three types of numbers and the implicit conversions among them are specified: naturals (that is, natural numbers) in Figure 5.8, integers in Figure 5.9, and floats (that is, floating point numbers) in Figure 5.10. The specification of integers given in Figure 5.9 is an alternative for the one in Figure 4.11 (which does not define integers in terms of naturals).

These examples instantiate the standard set of arithmetic operations that is discussed in Section 4.7. The binary function `.` (the dot) in `FloatM` can be used to construct the usual denotations for floats (for example, `2.20371`). Note that `.` is declared on numerals (see Section 3.7) rather than naturals because the leading zeros of the fractional part are significant (otherwise `1.05` would be equal to `1.5`).

Subsort relationships are defined to convert naturals to integers and integers to floats. The implicit function `Nat` convert numerals to naturals. This is not a subsorts relationship because `Nat` is not injective. The relationship can both be viewed as an is-a relationship (a numeral is a representation of a natural) or a has-a relationship (a numeral has a natural as value). Fortunately, the implicit function mechanism does not force one to choose.

This example is different from a typical object-oriented model like the shape example of the previous section. The three number sorts are defined inductively instead of listing key features. The inheritance relationships between the three sorts are, therefore, not declared by implicit key features. Object-oriented languages in general do not have inductive types. But, apart from that, in a object-oriented language c_1 can only be declared to be a subclass of c_2 as part of the declaration of c_1. In the example given here the situation is opposite: the "subclasses" are defined independently of their "superclasses". For example, `NatS` is made a subsort of `IntS` as part of the definition of `IntS`. This flexibility in defining inheritance relationships is possible because implicit functions are declared independently of sorts.

5.6 Example: Retract Functions

From an object-oriented point of view numeric operations such as `+` should be declared only once at the most general level (`FloatS`), and passed to more specific levels by inheritance. But, the addition of two numbers will then always be of type `FloatS`, even if the arguments are of type `IntS`. Terms like `5 Div (1+2)` will then

```
MODULE NatM

VAR    n   : NatS
VAR    num : NumeralS

IMPORT LogicM % Fig. 3.3 page 43

SORT   NatS (INDUCTIVE)
FUNC   ZeroNat :      -> NatS (CONSTRUCTOR)
FUNC   Succ    : NatS -> NatS (CONSTRUCTOR)

AXIOM  ZeroNat /= Succ n
AXIOM  Succ n1 /= Succ n2  <==  n1 /= n2

FUNC   Nat : NumeralS -> NatS (IMPLICIT)
AXIOM  Nat num = {the natural number represented by 'num'}

IMPORT PlusTimesM [NatS]

AXIOM  Zero = ZeroNat

AXIOM  Zero    + n  = n
AXIOM  Succ n1 + n2 = Succ (n1 + n2)

AXIOM  One = Succ Zero

AXIOM  Zero    * n  = Zero
AXIOM  Succ n1 * n2 = n1 * n2 + n2

%... more operations for naturals

END MODULE
```

Fig. 5.8: Specification of natural numbers.

```
MODULE Int2M

VAR    n : NatS
VAR    i : IntS

IMPORT NatM % Fig. 5.8 page 94

SORT   IntS (INDUCTIVE)
FUNC   Int : NatS -> IntS (CONSTRUCTOR)
FUNC   Min : NatS -> IntS (CONSTRUCTOR)

AXIOM  Int n1 /= Int n2  <==  n1 /= n2
AXIOM  Min n1 /= Min n2  <==  n1 /= n2
AXIOM  Int 0   = Min 0
AXIOM  Int n1 /= Min n2  <==  n1 /= 0  Or  n2 /= 0

IMPORT SubsortM [NatS, IntS] % Fig. 5.1 page 87

AXIOM  Inject n = Int n

IMPORT PlusMinusTimesM [IntS]

AXIOM  Zero:->IntS = Int Zero

AXIOM  Int n1      + Int n2      = Int (n1 + n2)
AXIOM  Min n1      + Min n2      = Min (n1 + n2)
AXIOM  Int Succ n1 + Min Succ n2 = Int n1 + Min n2
AXIOM  Min Succ n1 + Int Succ n2 = Min n1 + Int n2

AXIOM  -- Int n = Min n
AXIOM  -- Min n = Int n

AXIOM  Int n1 * Int n2 = Int (n1 * n2)
AXIOM  Min n1 * Min n2 = Int (n1 * n2)
AXIOM  Int n1 * Min n2 = Min (n1 * n2)
AXIOM  Min n1 * Int n2 = Min (n1 * n2)

%... more functions for integers such as Div and Mod

END MODULE
```

Fig. 5.9: Specification of integers. This specification is an alternative for the one given in Figure 4.11.

```
MODULE FloatM

VAR    i  : IntS
VAR    f  : FloatS
VAR    ds : NumeralS

IMPORT Int2M % Fig. 5.9 page 95

SORT   FloatS (INDUCTIVE)
FUNC   E : IntS, IntS -> FloatS (CONSTRUCTOR)

AXIOM  (i1 * 10) E i2 = i1 E (i2 + 1)
AXIOM  i1 /= i5 * 10  And  i3 /= i6 * 10  And
       i1 /= i3  And  i2 /= i4
       ==>
       i1 E i2 /= i3 E i4

IMPORT SubsortM [IntS, FloatS] % Fig. 5.1 page 87
AXIOM  Inject i = i E 0

FUNC   . : NumeralS, NumeralS -> FloatS
AXIOM  ds1.ds2
       = {the float with whole part 'ds1' and fractional part 'ds2'}

IMPORT PlusMinusTimesM [FloatS]

AXIOM  Zero:->FloatS = Zero E Zero

AXIOM  (i1 E i2) + (i3 E i2) = (i1 + i3) E i2

AXIOM  -- (i1 E i2) = (-- i1) E i2

AXIOM  (i1 E i2) * (i3 E i4) = (i1 * i3) E (i2 + i4)

FUNC   Pi : -> FloatS
AXIOM  Pi = 3.14

IMPORT TotalOrderedM [FloatS] % Fig. 4.7 page 75

%... definition of <= and other operations for FloatS

END MODULE
```

Fig. 5.10: Specification of floating point numbers.

```
MODULE Retract1M [XS, YS]

VAR    x : XS

SORT   XS
SORT   YS

IMPORT SubsortM [XS, YS] % Fig. 5.1 page 87

FUNC   Retract : YS -> XS (IMPLICIT)
AXIOM  Retract Inject x = x

END MODULE
```

Fig. 5.11: Specification of generic retract function.

be ill-typed (assuming the co-domain of `Div` is `IntS`). Typed object-oriented programming languages face the same problem; a solution some languages offer is the use of type-casting to make `1+2` officially a term of type `IntS` (for example, by writing `5 Div (1+2:IntS)`). It is possible to define an implicit type-cast operation, a so-called retract function (named after the similar concept of retracts in OBJ (Goguen et al. 1988)).

The *retract function* from a sort `XS` to a subsort `YS` is the inverse of the injection from `YS` to `XS`. The specification of retract functions is given in Figure 5.11. Strictly, `Retract` is a partial function because not all elements of `XS` have to be in the co-domain of `Inject`. Partiality is not discussed until Chapter 6. In Section 6.5 it is shown how the retract function can be specified as a partial function.

Using the retract function from `FloatS` to `IntS` (made available by replacing the import of `SubsortM` in module `FloatM` by `Retract1M`) is an alternative for having overloaded arithmetic operations for both floats and integers. For example, the retract function causes `5 Div (1+2)` to be well-typed because it then is an abbreviation of `5 Div (Retract (1+2))`. See Figure 5.11.

A disadvantage of defining the retract function from floats to integers is that all terms of type `FloatS` can then pass as integers, even if they do not represent an integer. The retract function causes `FloatS` to behave like an alias of `IntS`. Retract functions have to be used with care, otherwise the separation between sorts gets blurred and too many terms are well-typed.

5.7 Inheritance Ambiguity

The given examples contain a number of ambiguous assertions; that is, conversions can be inserted in more than one way. Declaration of implicit functions often results in ambiguous assertions, making it virtually impossible to write unambiguous specifications. However, these ambiguities are often harmless because either the assertion is semantically unambiguous (that is, all possible expansions have the same semantics) or there are good reasons to prefer one particular expansion over all others. In the current section a number of examples of inheritance ambiguities are given. In the next section a preference mechanism is presented that reduces inheritance ambiguity to an acceptable level.

The effect of inheritance on ambiguity is similar to that of overloading. For example, consider an implicit function f and a non-implicit function g such that:

$$f : s_1 \to s_2$$
$$g : s_2, s_3 \to s_4$$

then effectively there is a second "overloaded" version of g of type $s_1, s_3 \to s_4$. If there is also an overloaded version of g of type $s_1, s_3 \to s_4$, then effectively there are three overloaded versions of g of which two have the same type. There are two types of ambiguities that can be caused by inheritance: overloading conflicts and repeated inheritance.

5.7.1 Overloading Conflicts

Overloading conflicts occur when in a function application $f(\ldots)$ the overloaded abbreviation f can be completed in more than one way. For example, using the definitions of the number example, the term `0.5+(1+2)=2.5` is ambiguous since the numerals `1` and `2` can be summed using the overloaded `+` either for naturals, integers, or floats:

```
0.5 + (Float Int ((Nat 1) + (Nat 2))) = 3.5
0.5 + (Float ((Int Nat 1) + (Int Nat 2))) = 3.5
0.5 + ((Float Int Nat 1) + (Float Int Nat 2))) = 3.5
```

All of these expansions have the same semantics and, therefore, it is unsatisfactory to consider `0.5+(1+2)=3.5` ill-formed because of its syntactic ambiguity.

In general it is not true that different expansions have the same semantics. For example, using the definitions of the shape example, the equation `rect1=rect2` (where `rect1` and `rect2` are rectangle variables) has well-typed expansions (the first one without any conversions):

```
rect1 = rect2
(Shape rect1) = (Shape rect2)
(Position Shape rect1) = (Position Shape rect2)
```

These three expansions do not need to have the same semantics. Assume, for example, that `rect1` and `rect2` denote different rectangles that have the same area and position. Then the first expansion is false, but the other two are true. Two unequal rectangles can be equal as shapes because position and area are the key features of shapes, not of rectangles. Since a rectangle is a shape, this could be considered bad modeling practice and a reason to remove the equality axioms for `ShapeS`; adding an axiom for the injectivity of the implicit function `Shape` instead. The last expansion then still would have a different value than the first two because `Position` is not injective. This could be a reason to reject the use of inheritance for has-a relationships. Even if the ambiguities are not eliminated, the first expansion is the most obvious reading of `rect1=rect2` because it does not use any conversions. So, even if a term is semantically ambiguous there can be good reasons to prefer one particular expansion over the others.

An ambiguous term can be both semantically unambiguous and have a most obvious reading at the same time. For example, possible expansions of `1+2=3` are:

```
(Nat 1) + (Nat 2) = (Nat 3)
(Int Nat 1) + (Int Nat 2) = (Int Nat 3)
(Int ((Nat 1) + (Nat 2))) = (Int Nat 3)
(Float Int Nat 1) + (Float Int Nat 2) = (Float Nat 3)
(Float ((Int Nat 1) + (Int Nat 2))) = (Float Nat 3)
(Float Int ((Nat 1) + (Nat 2))) = (Float Nat 3)
```

All these expansions have the same semantics, but it is also pointless to insert conversions from naturals to integers or floats since the equation can already be evaluated at the level of naturals. Also, for `0.5+(1+2)`=3.5 it can be argued that the first expansion is the most obvious reading since there is no point in converting `1` and `2` to integers or floats before adding them.

5.7.2 Repeated Inheritance

The second kind of inheritance related ambiguity is caused by *repeated inheritance* by a sort s_1 from s_2. That is, if there are different sequences of implicit functions from s_1 to s_2.

The simplest case is *direct repeated inheritance* where there are implicit functions f_1 and f_2 both of type $s_1 \to s_2$. Then, for function $g : s_2 \to s_3$ and term $t : s_1$, the term $g(t)$ can be expanded to $g(f_1(t))$ and $g(f_2(t))$. In this situation there is no reason to favor one expansion over the other. But, there seems to be no reason to have multiple implicit functions from s_1 to s_2 in the first place, one is enough (that is why

no relevant example of this form of repeated inheritance is given here). So, in case of ambiguity caused by direct repeated inheritance it is acceptable to reject an assertion as being ill-formed.

A more complicated situation arises with *indirect repeated inheritance*, where $s_1 <^* x <^* s_2$ and $s_1 <^* y <^* s_2$ for distinct intermediate sorts x and y. Such a form of repeated inheritance, where the properties of x and y are combined in s_1, is similar to multiple inheritance in object-oriented languages. It can sometimes be useful; Meyer (1988) discusses repeated inheritance and gives the example of transcontinental drivers, which are drivers that drive cars on two different continents. An adaptation of this example is given in Figure 5.12 (here "France" and "US" refer to the places where cars are driven, not citizenship). This example is a bit farfetched, but it does illustrate the problem. Meyer notes that repeated inheritance does occur in practice, but not frequently. In `DriverM` the axiom

```
AXIOM  NrOfViolations fud
       = FranceViolations fud + USViolations fud
```

would be ambiguous without the definition of the additional operation `Driver`, because then `NrOfViolations fud` could be expanded in two ways:

```
NrOfViolations Driver France fud
NrOfViolations Driver US fud
```

The symmetry of the example prohibits that one expansion can be favored over the other. This is remedied by the additional conversion function `Driver` from `FranceUSDriverS` to `DriverS`. With this implicit function the expansion

```
NrOfViolations Driver fud
```

may be favored over the other two because it uses less implicit functions to convert an element of `FranceUSDriverS` to `DriverS`. Moreover, the axioms for `Driver fud` also guarantee semantic unambiguity. It is not an elegant solution, but it does do the job.

A special form of repeated inheritance is *repeated self inheritance*, that is, repeated inheritance between a sort s and s itself. By definition s inherits from itself through the empty chain of implicit functions. Thus, if there is at least one non-empty chain of implicit functions from s to s, then there is repeated self inheritance. This occurs, for example, in the definition of aliases (see Section 5.3) and retract functions (see Section 5.6).

Repeated self inheritance always causes ambiguous expansion if a term $t : s$ is used in an assertion since conversions from s to s can be applied to t an arbitrary number of times. However, such cyclic conversions are always redundant, therefore, expansions without them are the most obvious.

```
MODULE DriverM

VAR    fud : FranceUSDriverS

IMPORT Int2M % Fig. 5.9 page 95

SORT   DriverS
FUNC   Age            : DriverS -> IntS
FUNC   Address        : DriverS -> StringS
FUNC   NrOfViolations : DriverS -> IntS

SORT   FranceDriverS
FUNC   Driver           : FranceDriverS -> DriverS (IMPLICIT)
FUNC   FranceViolations : FranceDriverS -> IntS

SORT   USDriverS
FUNC   Driver       : USDriverS -> DriverS (IMPLICIT)
FUNC   USViolations : USDriverS -> IntS

SORT   FranceUSDriverS
FUNC   FranceDriver : FranceUSDriverS -> FranceDriverS (IMPLICIT)
FUNC   USDriver     : FranceUSDriverS -> USDriverS (IMPLICIT)
AXIOM  NrOfViolations fud = FranceViolations fud + USViolations fud

% Additional operation to prevent ambiguity
FUNC   Driver : FranceUSDriverS -> DriverS (IMPLICIT)
AXIOM  Driver fud = Driver FranceDriver fud
AXIOM  Driver fud = Driver USDriver fud

END MODULE
```

Fig. 5.12: Specification of transcontinental drivers.

5.8 Preferred Expansions

If an assertion is ambiguous solely because of overloading it is reasonable to reject it as being ill-formed. There is not much choice here: the different instances of overloaded functions are probably semantically unrelated. But even if they were related, it is hard for an automatic type-checker to verify that an ambiguous term is semantically unambiguous. On the syntactic side, there is no such thing as "most obvious reading" of a term with ambiguous overloading, because there is no measure to compare different expansions. Inheritance ambiguity is different in this respect. The previous section shows that sometimes the ambiguities are innocent, because either the possible expansions are semantically equivalent or ambiguity is caused by unnecessary conversions.

Therefore, a preference mechanism is introduced that, in case of innocent inheritance ambiguity, picks one of the expansions as *the* expansion. In case of a semantically unambiguous assertion an arbitrary expansion can be chosen. However, semantic unambiguity cannot be verified automatically by a type-checker (unless an automatic proof tool is used). In case of unnecessary conversions, it is feasible to choose the expansion that has the least number of conversions, because there is no point in adding redundant conversions. It will turn out that "least number of conversions" can be defined in such a way that semantically unambiguous assertions like the ones in the given examples, also have a preferred expansion. From an object-oriented point of view minimizing the number of conversions reflects the idea that in case of an overloaded method the most specific instance possible is used (that is, the instance belonging to the class closest to the class of the receiving object).

We can prefer expansions with a minimal number of conversions for each individual function argument. This works well in some cases: for example, the preferred expansions for `1+2=3` and `rect1=rect2` then are:

```
(Nat 1) + (Nat 2) = (Nat 3)
rect1 = rect2
```

However, none of the expansions of `0.5+(1+2)=3.5` has a minimum number of conversions for all individual arguments. The problem here is that conversion can be inserted at different levels, either for the arguments of `1+2` or for its result. Also minimizing the total number of conversions does not work. For example, expansions of `--2+2.5=0.5` with the same minimal overall number of conversions are:

```
(Float -- Int Nat 2) + 2.5 = 0.5
(-- Float Int Nat 2) + 2.5 = 0.5
```

Since treating all arguments within a term equal does not solve all harmless ambiguities, they will be handled in a particular order: expansion of arguments in a term is minimized in postorder (that is, relative to the parse tree of the term). Now the preferred expansion of `0.5+(1+2)=3.5` is:

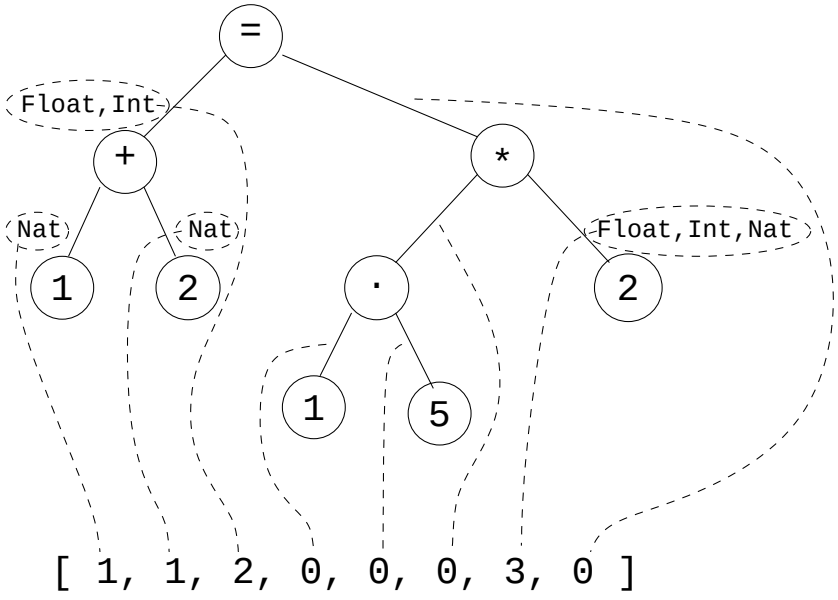

Fig. 5.13: The computation of the costs of the expansion of `1+2=1.5*2` to `Float Int (Nat 1 + Nat 2) = 1.5 * (Float Int Nat 2)`.

```
0.5 + (Float Int ((Nat 1) + (Nat 2)) = 3.5
```

With this form of preference the previous examples `1+2=3` and `rect1=rect2` still have the same preferred expansion.

This preference mechanism is defined by associating a cost with each expansion, $t \twoheadrightarrow t' : c$ denotes that t' is an expansion of t with costs c. Here c is a sequence of natural numbers, where each number represents the number of conversions added to one particular argument. That is, the cost of an expansion of $f(t_1, t_2, \ldots)$ first contains the cost of expanding t_1, then the number of conversions applied to t_1, then the cost for expanding t_2, the number of conversions applied to t_2, etcetera. An example of the computation of expansion costs is given in Figure 5.13. If a term has multiple expansions, the one with the minimum cost is preferred, where the costs are ordered lexically as lists. For example, the three expansions of `1+2=3` are (using abbreviated names):

$$\texttt{1+2=3} \twoheadrightarrow \texttt{(Nat 1) + (Nat 2) = (Nat 3)} : [1, 1, 0, 1]$$
$$\texttt{1+2=3} \twoheadrightarrow \texttt{(Int Nat 1) + (Int Nat 2) = (Int Nat 3)} : [2, 2, 0, 2]$$
$$\texttt{1+2=3} \twoheadrightarrow \texttt{(Int ((Nat 1) + (Nat 2))) = (Int Nat 3)} : [1, 1, 1, 2]$$

Here the preferred expansion is the first one with cost $[1, 1, 0, 1]$.

$$\frac{}{v \twoheadrightarrow v : [\,]}$$

$$\frac{\begin{array}{c} f \hookrightarrow f' \\ t_1 \twoheadrightarrow t_1' : c_1 \quad \cdots \quad t_n \twoheadrightarrow t_n' : c_n \\ t_1' \rightsquigarrow t_1'' : a_1 \quad \cdots \quad t_n' \rightsquigarrow t_n'' : a_n \end{array}}{f(t_1, \ldots, t_n) \twoheadrightarrow f'(t_1'', \ldots, t_n'') : c_1 +\!\!+ [a_1] +\!\!+ \ldots +\!\!+ c_n +\!\!+ [a_n]}$$

$$\frac{\begin{array}{c} s \hookrightarrow s' \\ t \twoheadrightarrow t' : c \end{array}}{(v : s \mid t) \twoheadrightarrow (v : s' \mid t') : c}$$

$$\frac{e_1 \rightarrowtail e_1' : c_1 \quad \cdots \quad e_n \rightarrowtail e_n' : c_n}{\{e_1 \ldots e_n\} \twoheadrightarrow \{e_1' \ldots e_n'\} : c_1 +\!\!+ \ldots +\!\!+ c_n}$$

$$\frac{}{nl \rightarrowtail nl : [\,]} \qquad \frac{nm \hookrightarrow nm'}{\text{'}nm\text{'} \rightarrowtail \text{'}nm'\text{'} : [\,]} \qquad \frac{t \twoheadrightarrow t' : c}{\text{'}t\text{'} \rightarrowtail \text{'}t'\text{'} : c}$$

$$\frac{}{t \rightsquigarrow t : 0} \qquad \frac{t \rightsquigarrow t' : a}{t \rightsquigarrow \pi(t') : a + 1}$$

Fig. 5.14: Definition of expansion rules, including implicit functions. For any implicit function π. (For any variable v; sorts s and s'; functions f and f'; terms $t_1, \ldots, t_n$, $t_1', \ldots, t_n'$, $t_1'', \ldots, t_n''$, t, and t'; integer lists $c_1, \ldots c_n$ and c; informals $e_1, \ldots e_n$ and $e_1', \ldots e_n'$; natural language nl; names nm and nm'; and integers $a_1, \ldots, a_n$, and a.)

The redefinition of the expansion relation $\twoheadrightarrow$ is given in Figure 5.14 (it replaces the definition of $\twoheadrightarrow$ in Figure 3.10). The auxiliary relation $\rightsquigarrow$ is used to add the actual conversions to function arguments: $t \rightsquigarrow t' : a$ denotes that t' is optained from t by applying a implicit functions to it. The operation $+\!\!+$ concatenates lists. Note that with respect to the calculation of the cost of an expansion, only function applications are relevant, variables, lambda-abstractions, and informal terms are ignored. The definition of $\twoheadrightarrow$ is extended further in Section 7.5 with so-called application redirection.